Managing Appliance Launches in Infrastructure Clouds

John Bresnahan
Mathematics and CS Division
Argonne National Laboratory
bresnahan@mcs.anl.gov

Tim Freeman
Computation Institute
University of Chicago
freeman@mcs.anl.gov

David LaBissoniere
Computation Institute
University of Chicago
labisso@uchicago.edu

Kate Keahey
Mathematics and CS Division
Argonne National Laboratory
Computation Institute
University of Chicago
keahey@mcs.anl.gov

ABSTRACT

Infrastructure cloud computing introduces a significant paradigm shift that has the potential to revolutionize how scientific computing is done. However, while it is actively adopted by a number of scientific communities, it is still lacking a well-developed and mature ecosystem that will allow the scientific community to better leverage the capabilities it offers. This paper introduces a specific addition to the infrastructure cloud ecosystem: the cloudinit.d program, a tool for launching, configuring, monitoring, and repairing a set of interdependent virtual machines in an Infrastructure-as-a-Service (IaaS) cloud or over a set of IaaS clouds. The cloudinit.d program was developed in the context of the Ocean Observatory Initiative (OOI) project to help it launch and maintain complex virtual platforms provisioned on-demand on top of infrastructure clouds. Like the UNIX init.d program, cloudinit.d can launch specified groups of services, and the VMs in which they run, at different run levels representing dependencies of the launched VMs. Once launched, cloudinit.d monitors the health of each running service to ensure that the overall application is operating properly. If a problem is detected in a service cloudinit.d will restart only that service, and any other service that failed which depended upon it.

General Terms

Management, Design, Experimentation.

Keywords

Cloud computing, Infrastructure-as-a-Service, Platform-as-a-Service, Nimbus.

1. INTRODUCTION

Infrastructure-as-a-service (IaaS) cloud computing [1] (sometimes also called "infrastructure cloud computing") has recently emerged as a promising outsourcing paradigm: it has been widely embraced commercially and is also beginning to make inroads in scientific communities. Infrastructure clouds allow users to exercise control over remote resources by introducing a virtualization layer that ensures isolation from the provider's infrastructure and thus a separation between provider's hardware and the user's environment. This feature proves particularly attractive to scientific communities where control over the environment is critical [2]. Furthermore, by providing on-demand access, cloud computing becomes an attractive solution to applications that are deadline-driven (e.g. experimental applications) or require urgent computing [3] capabilities.

Although many scientific projects are actively taking advantage of cloud computing, the development of its ecosystem is still in infancy. Tools enabling platform independent computing [4, 5], contextualization [6], elastic computing [7], or offering other functionality providing easy access to cloud facilities to the end user are still being developed. One particular need that emerged in this context is a tool enabling a controlled and repeatable launch and management of a set of virtual machines working in concert with each other to achieve a single goal. This task is often challenging as little can be assumed about the network locations of these virtual machines (their IP addresses are dynamically provisioned), they are frequently interdependent on each other, and their deployment can be spread across many different clouds providers, potentially supporting different interfaces. Specifically, the following questions arise: how can we orchestrate large-scale, multi-cloud, and multi-VM application launches? How can we organize, manage, and coordinate the bootstrap process of these complex cloud applications in a repeatable way? Once these applications are running how can we ensure that they continue to work and can we recover from failures without having to waste valuable time and potential data by completely restarting them?

In this paper, we introduce cloudinit.d, a tool for launching, configuring, monitoring, and repairing a set of interdependent virtual machines in an IaaS cloud or over a set of IaaS clouds. A single launch can consist of many VMs and can span multiple IaaS providers, including offerings from commercial and academic space like the many clouds offered by the FutureGrid project [14]. Like the UNIX, init.d program, cloudinit.d can launch specified groups of VMs at different run levels representing dependencies of the launched VMs. The launch is accomplished based on a well-defined launch plan. The launch plan is defined in a set of easy to understand text based *ini* formatted files that can be kept under version control. Because the launch process is repeatable, cloud applications can be developed in a structured and iterative way. Once launched, cloudinit.d monitors the health of each running service and the virtual machines in which they run. If a problem is detected in a service, cloudinit.d will restart only that service (and potentially it's hosting VM), and the dependencies of that service which also failed. Cloudinit.d was developed in the context of the Ocean Observatory Initiative project [8] to coordinate and repair launches of virtual machines of the common execution infrastructure of this project.

This paper is structured as follows. In Section 2 we define the requirements and design principles guiding the development of cloudinit.d. In Section 3 we describe its architecture and implementation. In Section 4 we present an application example and review the design features based on this example. In Section 5 we discuss related efforts. We conclude in Section 6.

2. REQUIREMENTS AND DESIGN PRINCIPLES

The guiding consideration of cloudinit.d development was to develop the simplest tool that could meet the following set of goals:

- The need to *provide repeatable, one-click,* deployment of sets of VMs. To achieve consistent behavior of systems, it is important to execute VM launches based on a launch plan that can be *created once and executed many times in exactly the same way*. The execution of this requirement is of course limited by the degree of repeatability provided by IaaS providers: in many cases it is impossible to repeat individual deployment actions (e.g., a deployment of an instance on the Amazon Web Services (AWS) provider [9] may result in many different instantiations [10]).

- Addressing a federated cloud deployment scenario. The cloudinit.d tool must be IaaS-agnostic so that it *can be deployed on any IaaS cloud*. To achieve portability and flexibility it is important to not only work with multiple providers but to launch different VMs on different clouds with a single launch plan. This can be achieved via the use of adapters, such as libcloud [5] or deltacloud [4] that provide a bridge to many IaaS cloud providers and services or by leveraging the increasing availability of standards such as the specification recently released by Open Grid Forum's Open Cloud Computing Interface (OCCI) working [11].

- *Coordination of interdependent launches.* The VMs within one launch can be interdependent in that information required for the deployment of one can be provided as a result of the deployment of another. For example, a VM may need to know the hostname of a database server to complete its launch sequence. On the other hand, VMs can also be independent and in this case can be deployed concurrently, without the need for coordination. Since it is important to accommodate both the interdependence and the optimization of concurrent launch of independent VMs, we divided the launch into *run levels* such that each level can define and resolve attributes to values that can be used by VMs launched in downstream run levels.

- *Testing and Reasoning* about the health of the system. To ensure repeatability, deal with complex launches and be able to reason about a complex system, a user needs to be able to make and verify assertions about vital properties of the system at any given time. Those assertions need to be both generic (e.g., "is the VM responding"?) and user-defined (e.g., testing an application-specific property of a system). For this reason it is important that user-defined launch tests embody assertions about the system. cloudinit.d should provide mechanisms that validate the correctness of a launch via user-defined tests run after the VMs have been deployed. To ensure meeting a wide range of useful tests they should be executed inside the VMs (e.g., based on ssh into the VM) rather then rely on external information only.

- *Ongoing monitoring of a launch.* To closely monitor the health of the system it is essential that the vital assertions about the system can be reevaluated at any time. Therefore, if such assertions are embedded in the launch tests, those tests need to be able to be rerun not just at launch, but at any time by an action triggered automatically or manually by the user (i.e., launch operator). It should be possible to store the results of monitoring tests in a database for launch analysis and recreation.

- *Policy-driven repair of a launch.* If any of the assertions about the system fail, it should be possible to repair the launch components by applying a repair action defined by a policy. For example, a failure can lead to a number of repeats of a launch action or abandonment of a launch component or even the whole launch if a component is deemed to be irreparable.

3. Architecture and Implementation

3.1 Launch Plans

Cloudinit.d arranges an application into three basic constructs: (1) service, (2) run level, (3) launch plan. These components are described below:

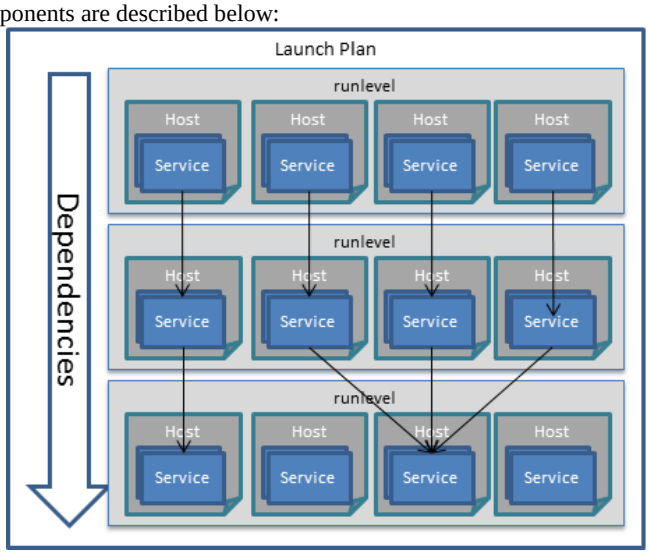

Figure 1: Launch plan example shows relationships between components: the first run-level contains all the services without dependencies as well as services that run-level 2 depends on; run-level 3 depends on run-level 2

- A *service* can be thought of as a single, configured VM. However, this is a very limiting definition. Many services can be configured to run in a single VM, or on an existing host that does not have to be a virtual machine at all. A service is an entity confined to a single machine which is responsible for a well defined task. In spite of this fact, in most of our examples we will merge the understanding of a single VM and a cloudinit.d service. Some example services are an HTTP server, a node in a Cassandra [12] pool, or a node in a RabbitMQ [13] message queue.

- A *run level* is a collection of services with no dependencies on each other. All services in a run level are launched at the same time. That run level launch is considered complete when all of the services in it have successfully started. Services in a run level can be run on one single cloud or across many different clouds. cloudinit.d makes no assumptions about locality. Any service in a run level can depend upon any service from a previous run level. For example, run level one forms a mongo DB data store cluster. A web application in run level 2 can depend on that mongo DB cluster, meaning, it can acquire all of the information needed to connect to it dynamically at boot time.
- A *launch* is an ordered set of run levels. To make a launch plan first all of the services are defined, then those services are arranged into run levels, and finally the run levels are put in a specific order. This forms a complete cloud (or inter-cloud) application.

Figure 1 shows how all of these components interact with each other. The arrows show the dependencies of one service on another. When a service needs information from another it depends upon it and thus must be in a higher run level. It can request dynamic information about another service at boot time, or repair time. This powerful feature allows the location of any given service to be entirely dynamic.

3.2 Services

There are two things that determine how a service behaves and how cloudinit.d interacts with it. Both of these are defined by the author of the launch plan (and thus the author of the service). The first is the base VM image (or running host). The software installed on that system and the software which is run automatically upon boot provides a baseline for the capabilities of the service. The second is the set of configuration scripts described below:

- The *bootpgm* is a program that is copied to a distinct location inside of the services host and is run once to setup the service. Often times it will download and install software, typically using tools like apt-get[15] or yum[16], and then configure that software for use. Tools like chef-solo[17] can be used by this script as well. The purpose of this program is to setup the host server with all needed software and start that software using any tools convenient to the user.

- The *readypgm* is similarly copied into the services host and run via ssh. This program's purpose is to check the status and health of the service. It can be, and typically is, run many times. As an example, if the service's goal was to serve HTTP, the readypgm would connect to localhost:80, download a known web page and check its content. If all is well the readypgm returns 0 and the service is reported as working. If not, the service is marked as *down* and the cloud application is in need of a repair.

- *terminatepgm* is the terminate program. It is run when a service is shutdown. It is there to nicely cleanup resources associated with the service.

3.3 Boot Process

Here we describe how a single service is brought into existence by cloudinit.d. A launch plan is given to the command line program which describes a cloud application with functionality arranged into the components describe above. The first thing cloudinit.d does is validate that the launch plan has no errors. Often times the acquisition of virtual machines is costly in terms of time and money, thus we want to avoid the case where the first 9 run levels pay the price only to discover that there was a bug in the launch plan in level 10. Once the plan is validated all IaaS requests for new virtual machines is made. We pre-stage this request as an optimization. Starting a new VM can take a significant amount of time and there are many tasks that we can do in parallel.

Cloudinit.d then starts monitoring the services at run level 1 waiting for the IaaS system on which the VM was launched to report an associated hostname. As soon as a hostname is present, cloudinit.d repeatedly, but not aggressively, attempts to ssh into the system. When a successful ssh connection is made the bootpgm is copied to a distinct location via scp. Additionally, a json[18] document which contains dependency information about all previously created services is copied to host. The bootpgm script is then run inside of that host with ssh. It can use the information in the json document for discovering values about its dependencies. As described above this script is responsible for setting up the service for use in the cloud application. If it is successful it must have an exit code of 0.

Along with the exit code, the bootpgm can return a json document. This document contains a set of key/value pairs which describe attributes of the newly configured service for consumption by higher level services. Cloudinit.d uses this document to furnish services at higher run levels with dynamic information about this newly created service.

Once all of the services at a run level have completed successfully, the process is then performed on the next run level until all run levels have completed.

3.4 Repair

In any distributed system failures are inevitable. In complex cloud applications diagnosing and repairing single failures can be an arduous task. cloudinit.d detects system failures by running the readypgm inside of the service's host machine. The user can manually decide when to check the status of their application with the cloudinit.d command line program or the cloudinit.d python API. If the readypgm returns a failure on a given service that service can be automatically repaired.

The repair process works by first running the terminatepgm (if it exists) and then shutting down the VM hosting the service. The boot process described above is then run on that service alone. In this reboot many of the dynamically determined attributes of this service (eg: hostname) will likely change. Because of this, once the reboot is complete, cloudinit.d will then test all the services at a higher level. If any of those services fail they too will be repaired in this way.

Conference'10, Month 1–2, 2010, City, State, Country.

4. APPLICATION EXAMPLE

Figure 2 shows an example cloud application run on four of the FutureGrid clouds using cloudinit.d. Here we have a highly available web application which uses a Cassandra distributed data base for highly available storage, apache HTTP servers for its web application, and a load balancer to distribute the work. To avoid a single point of failure and to ensure locality each node of both the Cassandra data base and the web farm are placed in different geographically distributed clouds. The load balancer is run on a single static host with a known location.

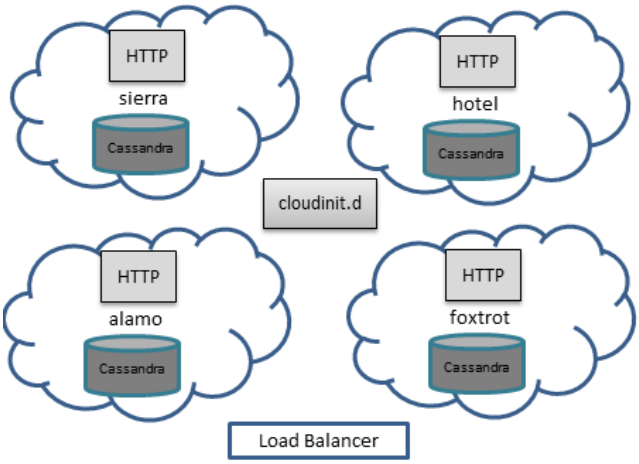

Figure 2: Example application using cloudinit.d deployed on FutureGrid Nimbus clouds

In this example each component is largely put in separate clouds for explanatory purposes. Whether or not this is the best architecture for all applications is outside of the scope of this discussion. Here we are presenting a reasonable real world example that helps illuminate important features of cloudinit.d.

4.1 Booting and Configuring a Single Instance

The creator of this application would write a launch plan with three run levels. The first has the four Cassandra nodes as services configured to run in each of the four FutureGrid clouds. The second is a set of replicated HTTP servers, configured similarly. The final run level is the load balancer which is run on a bare metal host that is assumed to be running prior to the boot of this application. The plan is configured in such a way as to route the important connection information from the Cassandra cluster, to each HTTP server. And similarly the list of HTTP servers is sent to the load balancer once run level 2 completes.

cloudinit.d takes a set of configuration files as input. Here we will introduce the reader to some of the details of the launch plan. This is not intended to be an exhaustive explanation of the details and syntax of the launch plan, but rather it is intended to provide the reader with a practical understanding of how it is intended to work. There are two main file types in a launch plan; a top level configuration file, and a run level configuration file. The top level configuration file simply enumerates the run levels and associates each run level with an additional configuration file. Our example top level configuration file follows:

```
[runlevels]
level1: cassandra.conf
level2: http.conf
level3: loadbalance.conf
```

This tells cloudinit.d that there are three run levels and where the description of those run levels can be found. Inside each of those files is a description of each service.

Here we will just introduce the *service* section of the boot level configuration file that describes the Cassandra service that will be run on the FutureGrid sierra cloud.

```
[svc-sierraCassandra]
iaas_key: XXXXXX
iaas_secret: XXXX
iaas_hostname: sierra.futuregrid.org
iaas_port: 8443
iaas: Nimbus

image: ubunut10.10
ssh_username: ubuntu
localsshkeypath: ~/.ssh/fg.pem
readypgm: cass-test.py
bootpgm: cass-boot.sh
```

The first five entries describe the cloud on which the service will be created. User security tokens and the contact point of the cloud are placed here. Cloudinit.d is also told what type of cloud has been described, in this case it *Nimbus* but it could be other common cloud types like EC2 [19] or Eucalyptus [20].

The line *image: ubuntu10.10* is a directive saying to request that the IaaS cloud launch the image with that name. In our example the image is a base Ubuntu 10.10 image. *ssh_username* and *localsshkeypath* give cloudinit.d the needed information for establishing a communication link with the VM instance launched from the *ubuntu10.10* image.

Readypgm and bootpgm point to scripts that perform the tasks associated with the respective directives (described in detail above). In our case cass-boot.sh will be copied to the VM instance with scp. The ssh session will be formed using the key at *localsshkeypath* and the username *ubuntu*. cass-boot.sh will then be run via ssh as the *ubuntu* user. It will download all of the software needed for Cassandra to run and configure this node to be a member of the four node cluster. When it is complete it will return back to cloudinit.d the contact information of the newly created Cassandra node. A similar entry is made in cassandra.conf for the other clouds.

Once all four Cassandra services have been successfully booted, cloudinit.d will open the configuration file http.conf. The contents of this file will look very similar to that of cassandra.conf, the main difference will be the bootpgm used to configure the system. Again four new VMs will be started on each of the 4 clouds. Cloudinit.d will again ssh into these machines and stage in the configuration scripts. However instead of setting up Cassandra, the bootgm will download, install, and configure a web server. Further it will connect to the Cassandra data store created in the previous boot level.

The final step in our example application is setting up the load balancer. In this case the host machine will not be a virtual machine. Instead it will be a static machine at a given hostname. The process for configuring it works in almost an identical way, only without the initial request to an IaaS framework to start a VM. That first step is skipped and the process continues by accessing the given hostname with scp and ssh. Because there are no further run levels, once this run level successfully completes details about each started service is reported to a log file and a summary is reported to the console for immediate observation by the operator.

5. RELATED WORK

CloudFormation [21] is a product created by Amazon Web Services. Much like cloudinit.d it is used to create and operate distributed applications in the cloud in a predictable and repeatable way. Unlike cloudinit.d CloudFormation cannot be used across many clouds. It is tool entirely dedicated for use with AWS only. Because of this it can take advantage of many of their services unavailable on other clouds (like SQS[22] and Elastic Beanstalk[23]), however it cannot be used with the vast resources available in science clouds. A further and important difference between the two systems is that cloudinit.d is designed to boot and contextualize an ordered hierarchy of VMs. CloudFormation is designed specifically to make all of the AWS services work in concert with each other, it does not do VM contextualization, and it does not have an explicit notion of boot order.

6. SUMMARY

This paper introduces cloudinit.d, a tool for launching, configuring, monitoring, and repairing a set of interdependent virtual machines in an Infrastructure-as-a-Service (IaaS) cloud or over a set of IaaS clouds. In addition, similar to its namesake, the UNIX init.d program, cloudinit.d can launch specified groups of VMs at different run levels representing dependencies of the launched VMs: this facilitates dealing with interdependencies with VM while optimizing the launch by allowing independent VMs to launch at the same time.

Cloudinit.d provides a new addition to the cloud computing ecosystem, making it easier for scientists to repeatedly launch, manage, and reason about sets of VMs in deployed the cloud. The capability to deploy launches repeatably is particularly important in the construction of stable system and the ability to evaluate, at any time, application-specific assertions significantly simplifies VM launches in cloud environment.

7. ACKNOWLEDGMENTS

This material is based on work supported in part by the National Science Foundation under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed." Partners in the FutureGrid project include U. Chicago, U. Florida, San Diego Supercomputer Center - UC San Diego, U. Southern California, U. Texas at Austin, U. Tennessee at Knoxville, U. of Virginia, Purdue I., and T-U. Dresden. This work also was supported in part by the Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. The OOI Cyberinfrastructure program is funded through the JOI Subaward, JSA 7-11, which is in turn funded by the NSF contract OCE-0418967 with the Consortium for Ocean Leadership, Inc.

8. REFERENCES

1. Armbrust, M., et al., Above the Clouds: A Berkeley View of Cloud Computing. 2009, University of California at Berkeley.

2. Keahey, K., T. Freeman, J. Lauret, and D. Olson. Virtual Workspaces for Scientific Applications. in SciDAC Conference. 2007. Boston, MA.

3. Beckman, P., S. Nadella, N. Trebon, and I. Beschastnikh, SPRUCE: A System for Supporting Urgent High-Performance Computing. IFIP International Federation for Information Process, Grid-Based Problems Solving Environments, 2007(239): p. 295-311.

4. deltacloud: http://incubator.apache.org/deltacloud/.

5. libcloud: a unified interface to the cloud: http://incubator.apache.org/libcloud/.

6. Keahey, K. and T. Freeman. Contextualization: Providing One-click Virtual Clusters. in eScience. 2008. Indianapolis, IN.

7. Marshall, P., K. Keahey, and T. Freeman, Elastic Site: Using Clouds to Elastically Extend Site Resources. CCGrid 2010, 2010.

8. Meisinger, M., C. Farcas, E. Farcas, C. Alexander, M. Arrott, J. de La Beaujardiere, P. Hubbard, R. Mendelssohn, and R. Signell. Serving Ocean Model Data on the Cloud. in Oceans 09. 2009.

9. Amazon Web Services (AWS): http://aws.amazon.com/.

10. Jackson, K., L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, and N. Wright. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud Amazon Web Services Cloud. in CloudCom. 2010.

11. Open Cloud Computing Interface (OCCI): http://occi-wg.org/.

12. Cassandra: http://cassandra.apache.org/.

13. RabbitMQ: http://www.rabbitmq.com/.

14. FutureGrid: https://portal.futuregrid.org/

15. Debian HowTo: http://www.debian.org/doc/manuals/apt-howto/

16. Fedora Project: http://fedoraproject.org/wiki/Tools/yum

17. OpsCode: http://wiki.opscode.com/display/chef/Chef+Solo

18. JSON: http://www.json.org/

19. EC2: http://aws.amazon.com/ec2/

20. Eucalyptus: http://www.eucalyptus.com/

21. CloudFormation: http://aws.amazon.com/cloudformation/

22. SQS: http://aws.amazon.com/sqs/

23. Elastic Beanstalk: http://aws.amazon.com/elasticbeanstalk/